Scientific software engineering with a simple ODE model as example

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Mar 21, 2015

Contents

Teaching material on scientific computing has traditionally been very on the mathematics and the applications, while details on how the com programmed to solve the problems have received little attention. Many writing as simple programs as possible and are not aware of much useful c science technology that would increase the fun, efficiency, and reliabilit their scientific computing activities.

This document demonstrates a series of good practices and tools from computer science, using a very simple mathematical problem with a ver implementation such that we minimize the mathematical details. Our g increase the technological quality of computer programming and make i the more well-established quality of the mathematics of scientific comp

More specifically we address the following scientific topics:

- How to structure a code in terms of functions
- How to make a module
- How to read input data flexibly from the command line
- How to create graphical/web user interfaces
- How to write unit tests (test functions or doctests)
- How to refactor code in terms of classes (instead of functions onl
- How to conduct and automate large-scale numerical experiments
- How to write scientific reports in various formats (LaTeX, HTML

The conventions and techniques outlined here will save you a lot of tin you incrementally extend software over time from simpler to more com problems. In particular, you will benefit from many good habits:

- new code is added in a modular fashion to a library (modules),
- programs are run through convenient user interfaces,

- it takes one quick command to let all your code can undergo heavy testing,
- tedious manual work with running programs is automated,
- your scientific investigations are reproducible,
- scientific reports with top quality typesetting are written both for paper and electronic devices.

Basic implementations

.1 Mathematical problem and solution technique

/e address the perhaps simplest possible differential equation problem

$$u'(t) = -au(t), \quad t \in (0, T], \tag{1}$$
$$u(0) = I, \tag{2}$$

here a, I, and T are prescribed parameters, and $u(t)$ is the unknown function › be estimated. This mathematical model is relevant for physical phenomena ›aturing exponential decay in time, e.g., vertical pressure variation in the ›tmosphere, cooling of an object, and radioactive decay.

The time domain is discretized with points $0 = t_0 < t_1 \cdots < t_{N_t} = T$, ›ere with a constant spacing Δt between the mesh points: $\Delta t = t_n - t_{n-1}$, $= 1, \ldots, N_t$. Let u^n be the numerical approximation to the exact solution at ›. A family of popular numerical methods can be written in the form

$$u^{n+1} = \frac{1 - (1-\theta)a\Delta t}{1 + \theta a\Delta t} u^n, \tag{3}$$

›r $n = 0, 1, \ldots, N_t - 1$. This numerical scheme corresponds to the Forward ›uler[1] scheme when $\theta = 0$, the Backward Euler[2] scheme when $\theta = 1$, and the ›rank-Nicolson[3] scheme when $\theta = 1/2$. The initial condition (2) is key to start ›e recursion with a value for u^0.

.2 A first, quick implementation

›olving (3) in a program is very straightforward: just make a loop over n and ›valuate the formula. The $u(t_n)$ values for discrete n can be stored in an array. ›his makes it easy to also plot the solution. We may with little work add the ›xact solution $u(t) = Ie^{-at}$ to the plot. A typical program is listed below.

[1] http://en.wikipedia.org/wiki/Forward_Euler_method
[2] http://en.wikipedia.org/wiki/Backward_Euler_method
[3] http://en.wikipedia.org/wiki/Crank-Nicolson


```
from numpy import *
from matplotlib.pyplot import *

A = 1
a = 2
T = 4
dt = 0.2
N = int(round(T/dt))
y = zeros(N+1)
t = linspace(0, T, N+1)
theta = 1
y[0] = A
for n in range(0, N):
    y[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*y[n]

y_e = A*exp(-a*t) - y
error = y_e - y
E = sqrt(dt*sum(error**2))
print 'Norm of the error: %.3E' % E
plot(t, y, 'r--o')
t_e = linspace(0, T, 1001)
y_e = A*exp(-a*t_e)
plot(t_e, y_e, 'b-')
legend(['numerical, theta=%g' % theta, 'exact'])
xlabel('t')
ylabel('y')
show()
```

This program is very easy to read, and as long it is correct, many w› that it has sufficient quality. Nevertheless, this program suffers from seri› habits that might be crucial for writing correct programs for more com› mathematical problems. First we list two really serious issues:

1. The notation in the program does not correspond exactly to the › in the mathematical problem: the solution is called `y` and corresp› u in the mathematical description, the variable `A` corresponds to t› ematical parameter I, `N` in the program is called N_t in the mathe›
2. There are no comments in the program.

Fixing notation and comments results in

```
from numpy import *
from matplotlib.pyplot import *

I = 1
a = 2
T = 4
dt = 0.2
Nt = int(round(T/dt))     # no of time intervals
u = zeros(Nt+1)           # array of u[n] values
t = linspace(0, T, Nt+1)  # time mesh
theta = 1                 # Backward Euler method

u[0] = I                  # assign initial condition
for n in range(0, Nt):    # n=0,1,...,Nt-1
    u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
```

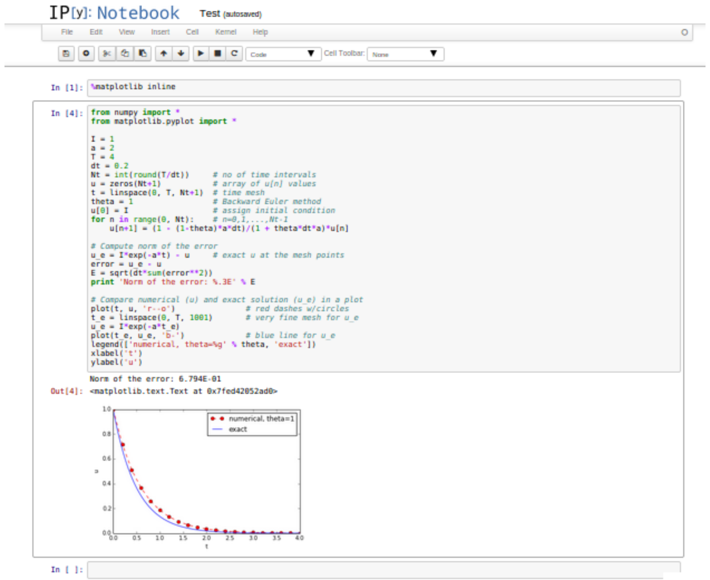


Figure 1: Flat experimental code in a notebook.

```
# Compute norm of the error
u_e = I*exp(-a*t) - u     # exact u at the mesh points
error = u_e - u
E = sqrt(dt*sum(error**2))
print 'Norm of the error: %.3E' % E

# Compare numerical (u) and exact solution (u_e) in a plot
plot(t, u, 'r--o')              # red dashes w/circles
t_e = linspace(0, T, 1001)      # very fine mesh for u_e
u_e = I*exp(-a*t_e)
plot(t_e, u_e, 'b-')            # blue line for u_e
legend(['numerical, theta=%g' % theta, 'exact'])
xlabel('t')
ylabel('u')
show()
```

At first sight, this is a good starting point for playing around with the xample: we can just change parameters and rerun. Let us embed the program ı an IPython notebook such that we can get the plot up in the notebook, see igure 1.

Although such an interactive session is good for initial exploration, one will oon extend the experiments and start developing the code further. Say we want › compare $\theta = 0, 1, 0.5$ in the same plot. This extension requires changes all ver the code and quickly lead to errors. To do something serious with this rogram we have to break it into smaller pieces and make sure each piece is ell tested, is general, and can be reused in new contexts without changes. The

next natural step is therefore to isolate the numerical computations a visualization in separate functions.

1.3 Isolating the numerical algorithm in a function

The solution formula (3) is completely general and should be availal Python function with all input data as function arguments and all outp returned to the calling code:

```
def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    dt = float(dt)               # avoid integer division
    Nt = int(round(T/dt))        # no of time intervals
    T = Nt*dt                    # adjust T to fit time step dt
    u = np.zeros(Nt+1)           # array of u[n] values
    t = np.linspace(0, T, Nt+1)  # time mesh

    u[0] = I                   # assign initial condition
    for n in range(0, Nt):     # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

Tip: Always use a doc string to document a function!

Python has a convention for documenting the purpose and usage function in a *doc string*: simply place the documentation in a on multi-line triple-quoted string right after the function header.

Be careful with unintended integer division!

Note that we in the `solver` function explicitly covert `dt` to a `float` o If not, the updating formula for `u[n+1]` may evaluate to zero becau integer division when `theta`, `a`, and `dt` are integers!

With the aid of the `solver` function, we can solve any problem of t (1)-(2) by a one-line statement:

```
u, t = solver(I=1, a=2, T=4, dt=0.2, theta=0.5)
```

One of the most serious flaws in computational work is to have several different implementations of the same computational algorithms lying in various program files. This is very likely to happen, because busy sc often want to test a slight variation of a code to see what happens. A qu and edit do the task, but such quick hacks have a tendency to survive a real correction is needed in the implementation, it is difficult to ensu ne correction is done in all relevant files. In fact, this is a general problem in rogramming, which has led to an important principle.

The DRY principle: Don't repeat yourself!

When implementing a particular functionality in a computer program, make sure this functionality and its variations are implemented in just one piece of code. That is, if you need to revise the implementation, there should be *one and only one* place to edit. It follows that you should never duplicate code (don't repeat yourself!), and code snippets that are similar should be factored into one piece (function) and parameterized (by function arguments).

.4 Making a module

s soon as you start making Python functions in a program, you should make re the program file fulfills the requirement of a module. This means that you an import and reuse your functions in other programs too. For example, if our olver function resides in a module `decay` in a module file `decay.py`, we can any program do

```
from decay import solver
# Solve a decay problem
u, t = solver(I=1, a=2, T=4, dt=0.2, theta=0.5)
```

r prefix function names by the module name:

```
import decay
# Solve a decay problem
u, t = decay.solver(I=1, a=2, T=4, dt=0.2, theta=0.5)
```

The requirements for a program to qualify for a module are simple:

1. The filename without `.py` must be a valid Python variable name.
2. The main program must be executed (through statements or a function call) in the *test block*.

he *test block* is normally placed at the end of a module file:

```
if __name__ == '__main__':
    # Statements
```

Vhen the module file is executed as a stand-alone program, the if test is true nd the indented statements are run, but when the module file is imported, `_name__` equals the module name and the test block is not executed.

To explain the importance of the test block, consider the trivial module file `ello.py` with one function and a call to this function as main program:


```
def hello(arg='World!'):
    print 'Hello, ' + arg

if __name__ == '__main__':
    hello()
```

Without the test block,

```
def hello(arg='World!'):
    print 'Hello, ' + arg

hello()
```

any attempt to import `hello` will also execute the call `hello()` and hen 'Hello, World!' to the screen. Such output is not desired when impo module! However, with the test block, `hello()` is not called during imp running the file as `python hello.py` will make the block active and lea desired printing.

All coming functions are placed in a module.

The many functions to be explained in the following text are put ir module file `decay.py`[a].

[a] `http://tinyurl.com/nm5587k/decay/decay.py`

What more than the `solver` function is needed in our `decay` modu everything we did in the previous, flat program? We need import sta for `numpy` and `matplotlib` as well as another function for producing t It can also be convenient to put the exact solution in a Python functio module `decay.py` then looks like this:

```
from numpy import *
from matplotlib.pyplot import *

def solver(I, a, T, dt, theta):
    ...

def exact_solution(t, I, a):
    return I*exp(-a*t)

def experiment_compare_numerical_and_exact():
    I = 1;  a = 2;  T = 4;  dt = 0.4;  theta = 1
    u, t = solver(I, a, T, dt, theta)

    t_e = linspace(0, T, 1001)       # very fine mesh for u_e
    u_e = exact_solution(t_e, I, a)

    plot(t,   u,   'r--o')           # dashed red line with circl
    plot(t_e, u_e, 'b-')             # blue line for u_e
    legend(['numerical, theta=%g' % theta, 'exact'])
    xlabel('t')
    ylabel('u')
```

```
    plotfile = 'tmp'
    savefig(plotfile + '.png');  savefig(plotfile + '.pdf')

    error = exact_solution(t, I, a) - u
    E = sqrt(dt*sum(error**2))
    print 'Error norm:', E

if __name__ == '__main__':
    experiment_compare_numerical_and_exact()
```

This module file does exactly the same as the previous, flat program, but becomes much easier to extend the code with other plots or experiments in ew functions. And even more important, the numerical algorithm is coded nd tested once and for all in the `solver` function, and any need to solve the athematical problem is a matter of one function call.

.5 Prefixing imported functions by the module name

nport statements of the form `from module import *` import functions and ariables in `module.py` into the current file. For example, when doing

```
from numpy import *
from matplotlib.pyplot import *
```

e get mathematical functions like `sin` and `exp` as well as MATLAB-style unctions like `linspace` and `plot`, which can be called by these well-known names. Infortunately, it sometimes becomes confusing to know where a particular unction comes from. Is it from `numpy`? Or `matplotlib.pyplot`? Or is it our wn function?

An alternative import is

```
import numpy
import matplotlib.pyplot
```

nd such imports require functions to be prefixed by the module name, e.g.,

```
t = numpy.linspace(0, T, Nt+1)
u_e = I*numpy.exp(-a*t)
matplotlib.pyplot.plot(t, u_e)
```

his is normally regarded as a better habit because it is explicitly stated from hich module a function comes from.

The modules `numpy` and `matplotlib.pyplot` are so frequently used, and heir full names quite tedious to write, so two standard abbreviations have volved in the Python scientific computing community:

```
import numpy as np
import matplotlib.pyplot as plt

t = np.linspace(0, T, Nt+1)
u_e = I*np.exp(-a*t)
plt.plot(t, u_e)
```


A version of the `decay_mod` module where we use the `np` and `plt` pr found in the file `decay_mod_prefix.py`[4].

The downside of prefixing functions by the module name is that mathe expressions like $e^{-at}\sin(2\pi t)$ get cluttered with module names,

```
numpy.exp(-a*t)*numpy.sin(2(numpy.pi*t)
# or
np.exp(-a*t)*np.sin(2*np.pi*t)
```

Such an expression looks like `exp(-a*t)*sin(2*pi*t)` in most other p ming languages. Similarly, `np.linspace` and `plt.plot` look less familiar ple who are used to MATLAB and who have not adopted Python's pre Whether to do `from module import *` or `import module` depends on p taste and the problem at hand. In these writings we use `from module` in more basic, shorter programs where similarity with MATLAB coul advantage. Prefix of mathematical functions in formulas is something w avoid to obtain a one-to-one correspondence between mathematical fo and the Python code.

Our `decay` module can be edited to use the module prefix for `matplotl` and `numpy`:

```
import numpy as np
import matplotlib.pyplot as plt

def solver(I, a, T, dt, theta):
    ...

def exact_solution(t, I, a):
    return I*np.exp(-a*t)

def experiment_compare_numerical_and_exact():
    I = 1;  a = 2;  T = 4;  dt = 0.4;  theta = 1
    u, t = solver(I, a, T, dt, theta)

    t_e = np.linspace(0, T, 1001)       # very fine mesh for u_e
    u_e = exact_solution(t_e, I, a)

    plt.plot(t,   u,   'r--o')        # dashed red line with circl
    plt.plot(t_e, u_e, 'b-')          # blue line for u_e
    plt.legend(['numerical, theta=%g' % theta, 'exact'])
    plt.xlabel('t')
    plt.ylabel('u')
    plotfile = 'tmp'
    plt.savefig(plotfile + '.png');  plt.savefig(plotfile + '.pdf

    error = exact_solution(t, I, a) - u
    E = np.sqrt(dt*np.sum(error**2))
    print 'Error norm:', E

if __name__ == '__main__':
    experiment_compare_numerical_and_exact()
```

[4] `http://tinyurl.com/nm5587k/softeng1/decay_mod_prefix.py`

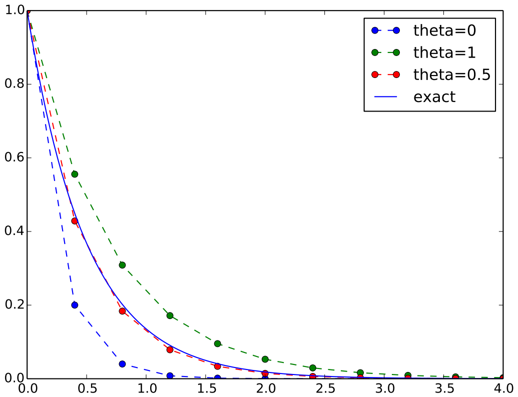
We remark that some would prefer to get rid of the prefix in mathematical ›rmulas:

```
from numpy import exp, sum, sqrt

def exact_solution(t, I, a):
    return I*exp(-a*t)

error = exact_solution(t, I, a) - u
E = sqrt(dt*sum(error**2))
```

.6 Comparing numerical schemes in one plot

et us specifically demonstrate one extension of the flat program in Section ref ıat would require substantial editing of the flat code, while in a structured ıodule, we can simply add a new function without affecting the existing code ıd with reusing the implementation of the numerics.

Our aim is to make a comparison between the numerical solutions for various :hemes (θ values) and the exact solution:



Stop a minute!

Look at the flat program in Section 1.6, and try to imagine which edits that are required to solve this new problem.

With the `solver` function at hand, we can simply create a function with a ›op over `theta` values and add the necessary plot statements:

```
def experiment_compare_schemes():
    """Compare theta=0,1,0.5 in the same plot."""
    I = 1;  a = 2;  T = 4;  dt = 0.4
```



```
    legends = []
    for theta in [0, 1, 0.5]:
        u, t = solver(I, a, T, dt, theta)
        plt.plot(t, u, '--o')            # dashed lines with circ
        legends.append('theta=%g' % theta)
    t_e = np.linspace(0, T, 1001)       # very fine mesh for u_e
    u_e = exact_solution(t_e, I, a)
    plt.plot(t_e, u_e, 'b-')            # blue line for u_e
    legends.append('exact')
    plt.legend(legends, loc='upper right')
    plotfile = 'tmp'
    plt.savefig(plotfile + '.png');  plt.savefig(plotfile + '.pdf
```

A call to this `experiment_compare_schemes` function must be place test block, or you can run the program from IPython instead:

```
In[1]: from decay import *

In[2]: experiment_compare_schemes()
```

2 User interfaces

It is good programming practice to let programs read input from the use than require the user to edit the source code when trying out new values parameters. One reason is that any edit of the code has a danger of intr bugs. Another reason is that it is easier and less manual work to supply a program instead of editing the program code. A third reason is that a p that reads input can easily be run by another program, and in this way automate a large number of runs in scientific investigations.

Reading input data can be done in many ways. We have to deci desired *user interface*, i.e., how we want to operate the program when p input, and then use appropriate tools to implement the user interface are four basic types of user interface of relevance to our programs, list with increasing complexity of the implementation:

1. Questions and answers in the terminal window
2. Command-line arguments
3. Reading data from file
4. Graphical user interfaces (GUIs)

Below, we shall address alternative 2 and 4, which are most appropriate present problem and also in general the most popular types of user inte

.1 Command-line arguments

he command-line arguments are all the words that appear on the command line fter the program name. Running a program `prog` as `prog arg1 arg2` means nat there are two command-line arguments (separated by white space): `arg1` and rg2. Python stores all the command-line arguments in the list `sys.argv`, and nere are, in principle, two ways of programming with command-line arguments ı Python:

- Decide upon a sequence of parameters on the command line and read their values directly from the `sys.argv[1:]` list (`sys.argv[0]` is the just program name).
- Use option-value pairs (`--option value`) on the command line to override default values of input parameters, and utilize the `argparse.ArgumentParser` tool to interact with the command line.

uppose we want to run our program `decay.py` with specification of a and Δt ı the command line. With positional command-line arguments we write

```
erminal> python decay.py 2 0.5
```

nowing that the first argument `2` is `a` (a) and the second '0.5'is `dt` (Δt).

With option-value pairs we can run

```
erminal> python decay.py --a 2 --dt 0.5
```

oth `a` and `dt` are supposed to have default values in the program, so we need › specify only the parameter that is to be changed from its default value, e.g.,

```
erminal> python decay.py --a 2      # a=2, default dt
erminal> python decay.py --dt 0.7   # dt-0.7, default a
erminal> python decay.py            # default a and dt
```

As example, we want to read I, a, T, θ, and a range of Δt values from the ommand line. A plot is then to be made, comparing the different numerical ılutions for different Δt values against the exact solution.

.2 Positional command-line arguments

he simplest way of reading the input parameters is to decide on their sequence ı the command line and just index the `sys.argv` list accordingly. Say the equence is I, a, T, θ followed by an arbitrary number of Δt values. This code xtract these *positional* command-line arguments:


```
import sys
I = float(sys.argv[1])
a = float(sys.argv[2])
T = float(sys.argv[3])
theta = float(sys.argv[4])
dt_values = [float(arg) for arg in sys.argv[5:]]
```

Command-line arguments are strings!

Note that all elements in `sys.argv` are string objects. If the values enter mathematical computations, we need to explicitly convert the st to numbers.

Instead of specifying the θ value, we could be a bit more sophisticated the user write the name of the scheme: `BE` for Backward Euler, `FE` for I Euler, and `CN` for Crank-Nicolson. Then we must map this string to the θ value, an operation elegantly done by a dictionary:

```
scheme = sys.argv[4]
scheme2theta = {'BE': 1, 'CN': 0.5, 'FE': 0}
if scheme in scheme2theta:
    theta = scheme2theta[scheme]
else:
    print 'Invalid scheme name:', scheme; sys.exit(1)
```

2.3 Option-value pairs on the command line

Now we want specify option-value pairs on the command line, using `-` (I), `--a` for `a` (a), `--T` for `T` (T), `--scheme` for the scheme name (`BE`, `FE`, (`--dt` for the sequence of `dt` (Δt) values. Each parameter must have a default value so that we specify the option on the command line only w default value is not suitable. Here is a typical run:

```
Terminal> python decay.py --I 2.5 --dt 0.1 0.2 0.01 --a 0.4
```

Observe the major advantage over positional command-line arguments: tl is much easier to read and much easier to write. With positional argume easy to mess up the sequence of the input parameters and quite challer detect errors too, unless there are just a couple of arguments.

Python's `ArgumentParser` tool in the `argparse` module makes it create a professional command-line interface to any program. The dc tation of `ArgumentParser`[5] demonstrates its versatile applications, so

[5]`http://docs.python.org/library/argparse.html`

ere just list an example containing basic features. It always pays off to use rgumentParser rather than trying to inspect and interpret sys.argv manually!

The use of ArgumentParser typically involves three steps:

```
import argparse
parser = argparse.ArgumentParser()

# Step 1: add arguments
parser.add_argument('--option_name', ...)

# Step 2: interpret the command line
args = parser.parse_args()

# Step 3: extract values
value = args.option_name
```

A function for setting up all the options is handy:

```
def define_command_line_options():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--I', '--initial_condition', type=float,
        default=1.0, help='initial condition, u(0)',
        metavar='I')
    parser.add_argument(
        '--a', type=float, default=1.0,
        help='coefficient in ODE', metavar='a')
    parser.add_argument(
        '--T', '--stop_time', type=float,
        default=1.0, help='end time of simulation',
        metavar='T')
    parser.add_argument(
        '--scheme', type=str, default='CN',
        help='FE, BE, or CN')
    parser.add_argument(
        '--dt', '--time_step_values', type=float,
        default=[1.0], help='time step values',
        metavar='dt', nargs='+', dest='dt_values')
    return parser
```

Each command-line option is defined through the parser.add_argument ıethod. Alternative options, like the short --I and the more explaining version -initial_condition can be defined. Other arguments are type for the Python bject type, a default value, and a help string, which gets printed if the command-ne argument -h or --help is included. The metavar argument specifies the alue associated with the option when the help string is printed. For example, ıe option for I has this help output:

```
erminal> python decay_argparse.py -h
 ...
 --I I, --initial_condition I
                       initial condition, u(0)
 ...
```


The structure of this output is

```
--I metavar, --initial_condition metavar
                      help-string
```

Finally, the --dt option demonstrates how to allow for more than o (separated by blanks) through the nargs='+' keyword argument. A command line is parsed, we get an object where the values of the opt stored as attributes. The attribute name is specified by the dist k argument, which for the --dt option is dt_values. Without the dest ar the value of an option --opt is stored as the attribute opt.

The code below demonstrates how to read the command line and ext values for each option:

```
def read_command_line_argparse():
    parser = define_command_line_options()
    args = parser.parse_args()
    scheme2theta = {'BE': 1, 'CN': 0.5, 'FE': 0}
    data = (args.I, args.a, args.T, scheme2theta[args.scheme],
            args.dt_values)
    return data
```

As seen, the values of the command-line options are available as at in args: args.opt holds the value of option --opt, unless we used t argument (as for --dt_values) for specifying the attribute name. The ar attribute has the object type specified by type (str by default).

The making of the plot is not dependent on whether we read data f command line as positional arguments or option-value pairs:

```
def experiment_compare_dt(option_value_pairs=False):
    I, a, T, theta, dt_values = \
       read_command_line_argparse() if option_value_pairs else \
       read_command_line_positional()

    legends = []
    for dt in dt_values:
        u, t = solver(I, a, T, dt, theta)
        plt.plot(t, u)
        legends.append('dt=%g' % dt)
    t_e = np.linspace(0, T, 1001)       # very fine mesh for u_e
    u_e = exact_solution(t_e, I, a)
    plt.plot(t_e, u_e, '--')            # dashed line for u_e
    legends.append('exact')
    plt.legend(legends, loc='upper right')
    plt.title('theta=%g' % theta)
    plotfile = 'tmp'
    plt.savefig(plotfile + '.png');  plt.savefig(plotfile + '.pdf
```

2.4 Creating a graphical web user interface

The Python package Parampool[6] can be used to automatically generat based *graphical user interface* (GUI) for our simulation program. A

[6] https://github.com/hplgit/parampool

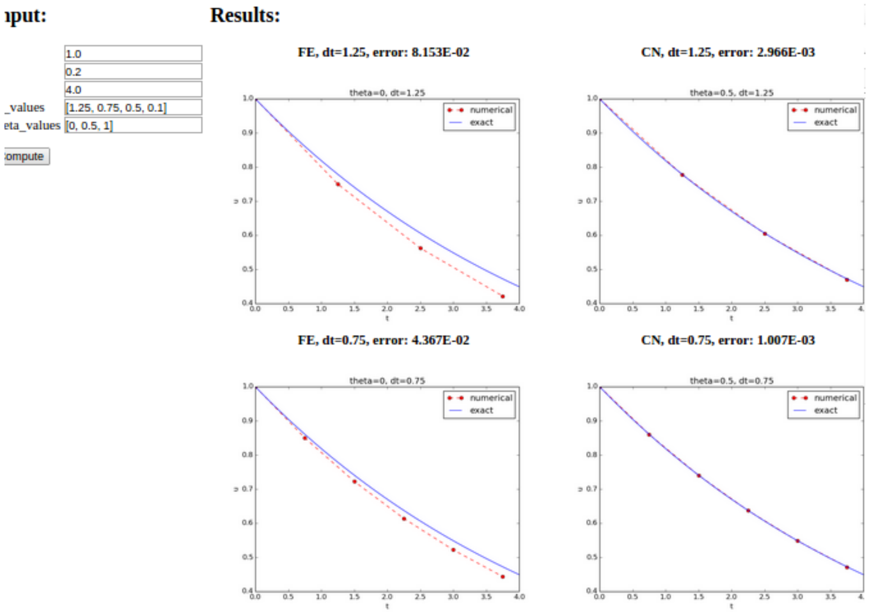


Figure 2: Automatically generated graphical web interface.

ıe programming technique dramatically simplifies the efforts to create a GUI, ıe forthcoming material on equipping our `decay_mod` module with a GUI is uite technical and of significantly less importance than knowing how to make a ommand-line interface.

Iaking a compute function. The first step is to identify a function that erforms the computations and that takes the necessary input variables as rguments. This is called the *compute function* in Parampool terminology. The urpose of this function is to take values of I, a, T together with a sequence of t values and a sequence of θ and plot the numerical against the exact solution ır each pair of $(\theta, \Delta t)$. The plots can be arranged as a table with the columns eing scheme type (θ value) and the rows reflecting the discretization parameter Δt value). Figure 2 displays of the graphical web interface may look like after esults are computed (there are 3×3 plots in the GUI, but only 2×2 are visible ı the figure).

To tell Parampool what type of input data we have, we assign default values f the right type to all arguments in the compute function, here called `main_GUI`:

```
lef main_GUI(I=1.0, a=.2, T=4.0,
             dt_values=[1.25, 0.75, 0.5, 0.1],
             theta_values=[0, 0.5, 1]):
```


The compute function must return the HTML code we want for dis the result in a web page. Here we want to show a table of plots. Assume that the HTML code for one plot and the value of the norm of the er be computed by some other function `compute4web`. The `main_GUI` func then loop over Δt and θ values and put each plot in an HTML table. App code goes like

```
def main_GUI(I=1.0, a=.2, T=4.0,
             dt_values=[1.25, 0.75, 0.5, 0.1],
             theta_values=[0, 0.5, 1]):
    # Build HTML code for web page. Arrange plots in columns
    # corresponding to the theta values, with dt down the rows
    theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
    html_text = '<table>\n'
    for dt in dt_values:
        html_text += '<tr>\n'
        for theta in theta_values:
            E, html = compute4web(I, a, T, dt, theta)
            html_text += """
<td>
<center><b>%s, dt=%g, error: %.3E</b></center><br>
%s
</td>
""" % (theta2name[theta], dt, E, html)
        html_text += '</tr>\n'
    html_text += '</table>\n'
    return html_text
```

Making one plot is done in `compute4web`. The statements should be : forward from earlier examples, but there is one new feature: we use a Parampool to embed the PNG code for a plot file directly in an HTMl tag. The details are hidden from the programmer, who can just rely on HTML code in the string `html_text`. The function looks like

```
def compute4web(I, a, T, dt, theta=0.5):
    """
    Run a case with the solver, compute error measure,
    and plot the numerical and exact solutions in a PNG
    plot whose data are embedded in an HTML image tag.
    """
    u, t = solver(I, a, T, dt, theta)
    u_e = exact_solution(t, I, a)
    e = u_e - u
    E = np.sqrt(dt*np.sum(e**2))

    plt.figure()
    t_e = np.linspace(0, T, 1001)    # fine mesh for u_e
    u_e = exact_solution(t_e, I, a)
    plt.plot(t,   u,   'r--o')       # red dashes w/circles
    plt.plot(t_e, u_e, 'b-')         # blue line for exact sol.
    plt.legend(['numerical', 'exact'])
    plt.xlabel('t')
    plt.ylabel('u')
    plt.title('theta=%g, dt=%g' % (theta, dt))
    # Save plot to HTML img tag with PNG code as embedded data
    from parampool.utils import save_png_to_str
```

```
    html_text = save_png_to_str(plt, plotwidth=400)

    return E, html_text
```

:enerating the user interface. The web GUI is automatically generated y the following code, placed in a file decay_GUI_generate.py[7]

```
from parampool.generator.flask import generate
from decay import main_GUI
generate(main_GUI,
         filename_controller='decay_GUI_controller.py',
         filename_template='decay_GUI_view.py',
         filename_model='decay_GUI_model.py')
```

unning the decay_GUI_generate.py program results in three new files whose ames are specified in the call to generate:

1. decay_GUI_model.py defines HTML widgets to be used to set input data in the web interface,
2. templates/decay_GUI_views.py defines the layout of the web page,
3. decay_GUI_controller.py runs the web application.

/e only need to run the last program, and there is no need to look into these les.

:unning the web application. The web GUI is started by

```
erminal> python decay_GUI_controller.py
```

)pen a web browser at the location 127.0.0.1:5000. Input fields for I, a, , dt_values, and theta_values are presented. Figure 2 shows a part of the esulting page if we run with the default values for the input parameters. With ne techniques demonstrated here, one can easily create a tailored web GUI for particular type of application and use it to interactively explore physical and umerical effects.

; Tests for verifying implementations

ny module with functions should have a set of tests that check the implemen-ations. There exists well-established procedures and corresponding tools for utomating the execution of such tests. One can in this way, with a one-line ommand, run large test sets and confirm that the software works (as far as the ests tells). Here we shall illustrate two important software testing techniques: *octest* and *unit testing*. The first one is Python specific, while unit testing is ne dominating test technique for computer software today.

[7] http://tinyurl.com/nm5587k/softeng1/decay_GUI_generate.py

3.1 Doctests

A doc string, the first string after the function header, is used to dc the purpose of functions and their arguments (see Section 1.3). Very is instructive to include an example on how to use the function. Int examples in the Python shell are most illustrative as we can see the resulting from function calls. For example, we can in the solver function an example on calling this function and printing the computed u and t

```
def solver(I, a, T, dt, theta):
    """
    Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt.


    >>> u, t = solver(I=0.8, a=1.2, T=1.5, dt=0.5, theta=0.5)
    >>> for t_n, u_n in zip(t, u):
    ...     print 't=%.1f, u=%.14f' % (t_n, u_n)
    t=0.0, u=0.80000000000000
    t=0.5, u=0.43076923076923
    t=1.0, u=0.23195266272189
    t=1.5, u=0.12489758761948
    """
    ...
```

When such interactive demonstrations are inserted in doc strings, P doctest[8] module can be used to automate running all commands in int sessions and compare new output with the output appearing in the do All we have to do in the current example is to run the module file decay.

```
Terminal> python -m doctest decay.py
```

This command imports the doctest module, which runs all doctests f the file.

Doctests prevent command-line arguments!

No additional command-line argument is allowed when running doc If your program relies on command-line input, make sure the tests ca run with such input.

The execution command above will report any problem if a test fa example, changing the last digit 8 in the output of the doctest to 7 tr report:

```
Terminal> python -m doctest decay.py
********************************************************
File "decay.py", line ...
Failed example:
```

[8] http://docs.python.org/library/doctest.html

```
    for t_n, u_n in zip(t, u):
        print 't=%.1f, u=%.14f' % (t_n, u_n)
xpected:
    t=0.0, u=0.80000000000000
    t=0.5, u=0.43076923076923
    t=1.0, u=0.23195266272189
    t=1.5, u=0.12489758761947
ot:
    t=0.0, u=0.80000000000000
    t=0.5, u=0.43076923076923
    t=1.0, u=0.23195266272189
    t=1.5, u=0.12489758761948
```

Pay attention to the number of digits in doctest results!

Note that in the output of `t` and `u` we write `u` with 14 digits. Writing all 16 digits is not a good idea: if the tests are run on different hardware, round-off errors might be different, and the `doctest` module detects that the numbers are not precisely the same and reports failures. In the present application, where $0 < u(t) \leq 0.8$, we expect round-off errors to be of size 10^{-16}, so comparing 15 digits would probably be reliable, but we compare 14 to be on the safe side. On the other hand, comparing a small number of digits may hide software errors.

Doctests are highly encouraged as they do two things: 1) demonstrate how a unction is used and 2) test that the function works.

Caution.

Doctests requires careful coding if they use command-line input or print results to the terminal window. Command-line input must be simulated by filling `sys.argv` correctly, e.g., `sys.argv = '--I 1.0 --a 5'.split`. The output lines of print statements inside doctests must be copied exactly as they appear when running the statements in an interactive Python shell.

.2 Unit tests and test functions

he unit testing technique consists of identifying small units of code, say a unction, and write one or more tests for each unit. One test should, ideally, not epend on the outcome of other tests. The recommended practice is actually to esign and write the unit tests first and *then* implement the functions!

In scientific computing it is not always obvious how to best perform unit esting. The units is naturally larger than in non-scientific software. Very often

the solution procedure of a mathematical problem identifies a unit, suc `solver` function.

Test frameworks: nose and pytest. Python offers two very eas software frameworks for implementing unit tests: nose and pytest. The (almost) in the same way, but my recommendation is to go for pytest.

Test function requirements. Each test can in these frameworks be as a *test function* that follows three rules:

1. The name must start with `test_`.
2. Function arguments are not allowed.
3. An `AssertionError` exception must be raised if the test fails.

A specific example might be illustrative before proceeding. Given a f that doubles the argument,

```
def double(n):
    return 2*n
```

a corresponding test function may look like this:

```
def test_double():
    """Test that double(n) works for one specific n."""
    n = 4
    expected = 2*4
    computed = double(4)
    if expected != computed:
        raise AssertionError
```

The last two lines, however, are never written like this in unit tests. use Python's `assert` statement: `assert success, msg`, where `succe` boolean variable, here `False` if the test fails, and `msg` is *an optional* string that is printed when the test fails. In detail, the test function lo

```
def test_double():
    """Test that double(n) works for one specific n."""
    n = 4
    expected = 2*4
    computed = double(4)
    msg = 'expected %g, computed %g' % (expected, computed)
    success = expected == computed
    assert success, msg
```

Comparison of real numbers. In scientific computing we very often deal with real numbers and round-off errors so the `==` operator must be by a comparison within a tolerance. Consider testing this function inst

```
lef third(x):
    return x/3.
```

e write a test function where the expected result is computed as $\frac{1}{3}x$ rather
ian $x/3$:

```
lef test_third():
    x = 0.1
    expected = (1/3.)*x
    computed = third(x)
    success = expected == computed
    assert success
```

his `test_third` function executes silently, i.e., no failure, for `x = 0.1`, but not
we set `x = 0.15`. The latter `x` value gives a round-off error. The solution to
iis problem is to compare `expected` and `computed` with a small tolerance:

```
lef test_third():
    x = 0.15
    expected = (1/3.)*x
    computed = third(x)
    tol = 1E-15
    success = abs(expected - computed) < tol
    assert success
```

pecial assert functions from nose. The nose test framework contains
iore tailored *assert functions* that can be called instead of using the `assert`
atement. For example, comparing two objects within a tolerance, as in the
resent case, can be done by `assert_almost_equal`:

```
import nose.tools as nt

lef test_third():
    x = 0.15
    expected = (1/3.)*x
    computed = third(x)
    nt.assert_almost_equal(
        expected, computed, delta=1E-15,
        msg='diff=%.17E' % (expected - computed))
```

ocating test functions. Test functions can reside in a module together with
ie functions they are supposed to verify, or the test functions can be collected
i separate files having names starting with `test`. Actually, nose and pytest
in automatically recursively run all test functions in all `test*.py` files in the
irrent and all subdirectories!

The `decay.py`[9] module file features test functions in the module, but we
ould equally well have made a subdirectory `tests` and put the test functions in
ay) `tests/test_decay.py`.

[9] `http://tinyurl.com/nm5587k/softeng1/decay.py`

Running tests. To run all test functions in the file `decay.py` do

```
Terminal> nosetests -s -v decay.py
Terminal> py.test -s -v decay.py
```

The `-s` option ensures that output from the test functions is printec
terminal window, and `-v` prints the outcome of each individual test fur

Alternatively, if the test functions are in some `test*.py` files, we
write `py.test -s -v` to *recursively* run *all* test functions in the current d
tree. The corresponding `nosetests -s -v` command does the same, but
subdirectory names to start with `test` or end with `_test` or `_tests` (wl
good habit anyway). An example of a `tests` directory with a `test*.p`
found in src/softeng1/tests[10].

Installing nose and pytest. With `pip` available, it is trivial to inst
and/or pytest: `sudo pip install nose` and `sudo pip install pyte`

3.3 Test function for the solver

Finding good test problems for verifying the implementation of numeric
ods is a topic on its own. The challenge is that we very seldom know w
numerical errors are. For the present model problem (1)-(2) solved by
can, fortunately, derive a formula for the numerical approximation:

$$u^n = I\left(\frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}\right)^n .$$

Then we know that the implementation should produce numbers that ag
this formula to machine precision. The formula for u^n is known as a
discrete solution of the problem:

```
def exact_discrete_solution(n, I, a, theta, dt):
    """Return exact discrete solution of the numerical schemes.""
    dt = float(dt)  # avoid integer division
    A = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
    return I*A**n
```

A test function can evaluate this solution on a time mesh and check th
values produced by the `solver` function do not deviate with more than
tolerance:

```
def test_exact_discrete_solution():
    """Check that solver reproduces the exact discr. sol."""
    theta = 0.8; a = 2; I = 0.1; dt = 0.8
    Nt = int(8/dt)  # no of steps
    u, t = solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)
```

[10] `http://tinyurl.com/nm5587k/softeng1/tests`

```
    # Evaluate exact discrete solution on the mesh
    u_de = np.array([exact_discrete_solution(n, I, a, theta, dt)
                     for n in range(Nt+1)])
    diff = np.abs(u_de - u).max()  # largest deviation
    tol = 1E-14
    success = diff < tol
    assert success
```

An important topic to address in test functions is potentially problematic ıput to functions. For example, if a, Δt, and θ are integers, one may face roblems with unintended integer division in the numerical solution algorithm ɪr the present mathematical problem. We should therefore add a test to make ıre our `solver` function does not fall into this potential trap:

```
lef test_potential_integer_division():
    """Choose variables that can trigger integer division."""
    theta = 1; a = 1; I = 1; dt = 2
    Nt = 4
    u, t = solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)
    u_de = np.array([exact_discrete_solution(n, I, a, theta, dt)
                     for n in range(Nt+1)])
    diff = np.abs(u_de - u).max()
    assert diff < 1E-14
```

.4 Test function for reading positional command-line arguments

he function `read_command_line_positional` extracts numbers from the comıand line. To test it, decide on a set of numbers, fill `sys.argv` appropriately, ıd check that we get the expected numbers:

```
lef test_read_command_line_positional():
    # Decide on a data set of input parameters
    I = 1.6;  a = 1.8;  T = 2.2;  theta = 0.5
    dt_values = [0.1, 0.2, 0.05]
    # Expected return from read_command_line_positional
    expected = [I, a, T, theta, dt_values]
    # Construct corresponding sys.argv array
    sys.argv = [sys.argv[0], str(I), str(a), str(T), 'CN'] + \
               [str(dt) for dt in dt_values]
    computed = read_command_line_positional()
    for expected_arg, computed_arg in zip(expected, computed):
        assert expected_arg == computed_arg
```

ote that `sys.argv[0]` is always the program name and that we have to copy ıat string from the original `sys.argv` array to the new one we construct in the ɛst function. (Actually, the test function destroys the original `sys.argv` that ython fetched from the command line.)

Any numerical code writer should always be skeptical to the use of the exact quality operator `==` in test functions, since round-off errors often come into play. ere, however, we set some real values, convert them to strings and convert back gain to real numbers (of the same precision). This string-number conversion

does not involve approximation so we can safely use `==` in tests. Note a the last element in `expected` and `computed` is the list `dt_values`, and = for comparing two lists too.

3.5 Test function for reading option-value pairs

Testing the function `read_command_line_argparse` follows the set up similar function for positional command-line arguments. However, the c tion of the command line is a bit more complicated. We find it conve construct the line as a string and then split the line into words to get the list `sys.argv`:

```
def test_read_command_line_argparse():
    I = 1.6;  a = 1.8;  T = 2.2;  theta = 0.5
    dt_values = [0.1, 0.2, 0.05]
    # Expected return from read_command_line_positional
    expected = [I, a, T, theta, dt_values]
    # Construct corresponding sys.argv array
    cml = '%s --a %s --I %s --T %s --scheme CN --dt ' % \
          (sys.argv[0], a, I, T)
    cml = cml + ' '.join([str(dt) for dt in dt_values])
    sys.argv = cml.split()
    computed = read_command_line_argparse()
    for expected_arg, computed_arg in zip(expected, computed):
        assert expected_arg == computed_arg
```

Let silent test functions speak up during development!

When you develop test functions in a module, it is common to use IPy to reload the module and call the test function as it gets developed:

```
In[1]: import decay

In[2]: decay.test_read_command_line_argparse()

In[3]: reload(decay)  # force new import

In[2]: decay.test_read_command_line_argparse()  # test again
```

However, a working test function is completely silent! Many find it ps logically annoying to convince themselves that a completely silent fun is doing the right things. It can therefore, during development of a function, be convenient to insert print statements in the function to mo that the function body is indeed executed... For example, one can the expected and computed values in the terminal window.

.6 Classical class-based unit testing

he test functions written for the nose and pytest frameworks are very straight-
orward and to the point, with no framework-required boilerplate code. We just
rite the statements we need to make the computations and comparisons and
nen make the require `assert`.

The classical way implementing unit tests derives from the JUnit tool in
ava and leads to much more comprehensive implementations with much more
oilerplate code. Python comes with a built-in module `unittest` for doing this
ype of unit tests. Although I strongly recommend to use nose or pytest over
nittest, class-based unit testing in the style of `unittest` has a very strong
osition in computer science and is so widespread that even computational
cientists should have an idea how such unit test code is written. A short demo
f `unittest` is therefore included next.

Suppose we have a function `double(x)` in a module file `mymod.py`:

```
def double(x):
    return 2*x
```

nit testing with the aid of the `unittest` module consists of writing a file
est_mymod.py for testing the functions in `mymod.py`. The individual tests
nust be methods with names starting with `test_` in a class derived from class
estCase in `unittest`. With one test method for the function `double`, the
est_mymod.py file becomes

```
import unittest
import mymod

class TestMyCode(unittest.TestCase):
    def test_double(self):
        x = 4
        expected = 2*x
        computed = mymod.double(x)
        self.assertEqual(expected, computed)

if __name__ == '__main__':
    unittest.main()
```

he test is run by executing the test file `test_mymod.py` as a standard Python
rogram. There is no support in `unittest` for automatically locating and
unning all tests in all test files in a directory tree.

We could use the basic `assert` statement as we did with nose and pytest
unctions, but those who write code based on `unittest` almost exclusively use the
ide range of built-in assert functions such as `assertEqual`, `assertNotEqual`,
ssertAlmostEqual, to mention some of them.

Translation of `test_exact_discrete_solution`, `test_potential_integer_div`
nd the other test functions in `decay.py` to `unittest` means making a new file
est_decay.py file with a test class `TestDecay` where the stand-alone functions
or nose/pytest now become methods in this class.


```
import unittest
import decay
import numpy as np

def exact_discrete_solution(n, I, a, theta, dt):
    ...

class TestDecay(unittest.TestCase):

    def test_exact_discrete_solution(self):
        theta = 0.8; a = 2; I = 0.1; dt = 0.8
        Nt = int(8/dt)  # no of steps
        u, t = decay.solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta
        # Evaluate exact discrete solution on the mesh
        u_de = np.array([exact_discrete_solution(n, I, a, theta,
                         for n in range(Nt+1)])
        diff = np.abs(u_de - u).max()  # largest deviation
        self.assertAlmostEqual(diff, 0, delta=1E-14)

    def test_potential_integer_division(self):
        ...
        self.assertAlmostEqual(diff, 0, delta=1E-14)

    def test_read_command_line_positional(self):
        ...
        for expected_arg, computed_arg in zip(expected, computed)
            self.assertEqual(expected_arg, computed_arg)

    def test_read_command_line_argparse(self):
        ...

if __name__ == '__main__':
    unittest.main()
```

4 Classes for problem and solution methoc

The θ-rule was compactly and conveniently implemented in a function sc
Section 1.1. In more complicated problems it might be beneficial to use
and introduce a class `Problem` to hold the definition of the physical p
and a class `Solver` to hold the data and methods needed to numerical
the problem. This idea will now be illustrated, resulting in code that re
an alternative to the `solver` and `experiment_*` functions found in th
module.

Explaining the details of class programming in Python is considered
the scope of this text. Readers who are unfamiliar with Python class progr
should first consult one of the many electronic Python tutorials or te
to come up to speed with concepts and syntax of Python classes before
on. The author has a gentle introduction to class programming for s
applications in [1], see Chapter 7 and 9 and Appendix E. Other useful r
are

- The Python Tutorial: `http://docs.python.org/2/tutorial/c`
  `html`

- Wiki book on Python Programming: http://en.wikibooks.org/wiki/Python_Programming/Classes
- tutorialspoint.com: http://www.tutorialspoint.com/python/python_classes_objects.htm

.1 The problem class

he purpose of the problem class is to store all information about the mathe- iatical model. This usually means the physical parameters and formulas in the roblem. Looking at our model problem (1)-(2), the physical data cover I, a, nd T. Since we have an analytical solution of the ODE problem, we may add iis solution in terms of a Python function (or method) to the problem class as ell. A possible problem class is therefore

```
from numpy import exp

class Problem:
    def __init__(self, I=1, a=1, T=10):
        self.T, self.I, self.a = I, float(a), T

    def u_exact(self, t):
        I, a = self.I, self.a
        return I*exp(-a*t)
```

/e could in the `u_exact` method have written `self.I*exp(-self.a*t)`, but sing local variables `I` and `a` allows the formula `I*exp(-a*t)` which looks closer › the mathematical expression Ie^{-at}. This is not an important issue with the ırrent compact formula, but is beneficial in more complicated problems with nger formulas to obtain the closest possible relationship between code and iathematics. My coding style is to strip off the `self` prefix when the code xpresses mathematical formulas.

The class data can be set either as arguments in the constructor or at any me later, e.g.,

```
problem = Problem(T=5)
problem.T = 8
problem.dt = 1.5
```

Some programmers prefer `set` and `get` functions for setting and getting data in asses, often implemented via *properties* in Python, but I consider that overkill hen we just have a few data items in a class.)

It would be convenient if class `Problem` could also initialize the data from the ommand line. To this end, we add a method for defining a set of command-line ptions and a method that sets the local attributes equal to what was found on ie command line. The default values associated with the command-line options re taken as the values provided to the constructor. Class `Problem` now becomes


```
class Problem:
    def __init__(self, I=1, a=1, T=10):
        self.T, self.I, self.a = I, float(a), T

    def define_command_line_options(self, parser=None):
        """Return updated (parser) or new ArgumentParser object."
        if parser is None:
            import argparse
            parser = argparse.ArgumentParser()

        parser.add_argument(
            '--I', '--initial_condition', type=float,
            default=1.0, help='initial condition, u(0)',
            metavar='I')
        parser.add_argument(
            '--a', type=float, default=1.0,
            help='coefficient in ODE', metavar='a')
        parser.add_argument(
            '--T', '--stop_time', type=float,
            default=1.0, help='end time of simulation',
            metavar='T')
        return parser

    def init_from_command_line(self, args):
        """Load attributes from ArgumentParser into instance."""
        self.I, self.a, self.T = args.I, args.a, args.T

    def u_exact(self, t):
        """Return the exact solution u(t)=I*exp(-a*t)."""
        I, a = self.I, self.a
        return I*exp(-a*t)
```

Observe that if the user already has an `ArgumentParser` object it can be s but if she does not have any, class `Problem` makes one. Python's `None` c used to indicate that a variable is not initialized with a proper value.

4.2 The solver class

The solver class stores data related to the numerical solution method and p a function `solve` for solving the problem. A problem object must be give constructor so that the solver can easily look up physical data. In the example, the data related to the numerical solution method consists of θ. We add, as in the problem class, functionality for reading Δt and θ f command line:

```
class Solver:
    def __init__(self, problem, dt=0.1, theta=0.5):
        self.problem = problem
        self.dt, self.theta = float(dt), theta

    def define_command_line_options(self, parser):
        """Return updated (parser) or new ArgumentParser object."
        parser.add_argument(
            '--scheme', type=str, default='CN',
            help='FE, BE, or CN')
        parser.add_argument(
```

```
            '--dt', '--time_step_values', type=float,
            default=[1.0], help='time step values',
            metavar='dt', nargs='+', dest='dt_values')
        return parser

    def init_from_command_line(self, args):
        """Load attributes from ArgumentParser into instance."""
        self.dt, self.theta = args.dt, args.theta

    def solve(self):
        self.u, self.t = solver(
            self.problem.I, self.problem.a, self.problem.T,
            self.dt, self.theta)

    def error(self):
        """Return norm of error at the mesh points."""
        u_e = self.problem.u_exact(self.t)
        e = u_e - self.u
        E = sqrt(self.dt*sum(e**2))
        return E
```

'ote that the solve method is just a *wrapper* of the previously developed ;and-alone solver function.

:ombining the objects. Eventually we need to show how the classes Problem ıd Solver play together:

```
lef experiment_classes():
    problem = Problem()
    solver = Solver(problem)

    # Read input from the command line
    parser = problem.define_command_line_options()
    parser = solver. define_command_line_options(parser)
    args = parser.parse_args()
    problem.init_from_command_line(args)
    solver. init_from_command_line(args)

    # Solve and plot
    solver.solve()
    import matplotlib.pyplot as plt
    t_e = np.linspace(0, T, 1001)    # very fine mesh for u_e
    u_e = problem.u_exact(t_e)

    plt.plot(t,   u,   'r--o')       # dashed red line with circles
    plt.plot(t_e, u_e, 'b-')         # blue line for u_e
    plt.legend(['numerical, theta=%g' % theta, 'exact'])
    plt.xlabel('t')
    plt.ylabel('u')
    plotfile = 'tmp'
    plt.savefig(plotfile + '.png');  plt.savefig(plotfile + '.pdf')

    error = problem.u_exact(t) - u
    E = np.sqrt(dt*np.sum(error**2))
    print 'Error norm:', E
    plt.show()
```



```
if __name__ == '__main__':
    experiment_compare_dt(True)
    plt.show()
```

4.3 Improving the problem and solver classes

The previous Problem and Solver classes containing parameters soon g repetitive code when the number of parameters increases. Much of this c be parameterized and be made more compact. For this purpose, we d collect all parameters in a dictionary, self.prm, with two associated dict self.type and self.help for holding associated object types and help For the specific ODE example we deal with, such dictionaries are

```
# Problem data
self.prm  = dict(I=1, a=1, T=10)
self.type = dict(I=float, a=float, T=float)
self.help = dict(I='initial condition, u(0)',
                 a='coefficient in ODE',
                 T='end time of simulation')
# Solver data
self.prm  = dict(dt=0.5, theta=0.5)
self.type = dict(dt=float, theta=float)
self.help = dict(dt='time step value',
                 theta='time discretization parameter')
```

Provided a problem or solver class defines these three dictionaries in structor, using default or user-supplied values of the parameters, we ca a super class Parameters with general code for defining command-line and reading them as well as methods for setting and getting a param Problem or Solver for a particular mathematical problem can then inhe of the needed functionality and code from the Parameters class.

A generic class for parameters. A simplified version of the parame looks as follows:

```
class Parameters:
    def set(self, **parameters):
        for name in parameters:
            if name in self.prm:
                self.prm[name] = parameters[name]
            else:
                raise NameError('Illegal parameter name %s' % nam

    def get(self, name):
        """Return value of parameter with given name."""
        return self.prm[name]

    def define_command_line_options(self, parser=None):
        """Automatic registering of options."""
        if parser is None:
            import argparse
            parser = argparse.ArgumentParser()
```

```
        for name in self.prm:
            tp = self.type[name] if name in self.type else str
            help = self.help[name] if name in self.help else None
            parser.add_argument(
                '--' + name, default=self.get(name), metavar=name,
                type=tp, help=help)

        return parser

    def init_from_command_line(self, args):
        for name in self.prm:
            self.prm[name] = getattr(args, name)
```

he file decay_oo.py[11] contains a slightly more advanced version of class arameters where we in the set and get functions test for valid parameer names and raise exceptions with informative messages if any name is not egistered.

he problem class. A class Problem for the problem $u' = -au$, $u(0) = I$, $\in (0, T]$, with parameters input a, I, and T can now be coded as

```
lass Problem(Parameters):
    """
    Physical parameters for the problem u'=-a*u, u(0)=I,
    with t in [0,T].
    """
    def __init__(self):
        self.prm  = dict(I=1, a=1, T=10)
        self.type = dict(I=float, a=float, T=float)
        self.help = dict(I='initial condition, u(0)',
                         a='coefficient in ODE',
                         T='end time of simulation')

    def u_exact(self, t):
        I, a = self.get('I'), self.get('a')
        return I*np.exp(-a*t)
```

he solver class. Also the solver class is derived from class Parameters and orks with the prms, types, and help dictionaries in the same way as class roblem. Otherwise, the code is very similar to the previous class Solver:

```
lass Solver(Parameters):
    def __init__(self, problem):
        self.problem = problem
        self.prm  = dict(dt=0.5, theta=0.5)
        self.type = dict(dt=float, theta=float)
        self.help = dict(dt='time step value',
                         theta='time discretization parameter')

    def solve(self):
        from decay_mod import solver
        self.u, self.t = solver(
```

[11] http://tinyurl.com/nm5587k/softeng1/decay_oo.py


```
            self.problem.get('I'),
            self.problem.get('a'),
            self.problem.get('T'),
            self.get('dt'),
            self.get('theta'))

    def error(self):
        try:
            u_e = self.problem.u_exact(self.t)
            e = u_e - self.u
            E = np.sqrt(self.get('dt')*np.sum(e**2))
        except AttributeError:
            E = None
        return E
```

The advantage with the Parameters class is that it scales to proble a large number of physical and numerical parameters: as long as the par are defined once via a dictionary, the compact code in class Paramet handle any collection of parameters of any size.

5 Automating scientific experiments

Empirical scientific investigations based on running computer programs careful design of the experiments and accurate reporting of results. A there is a strong tradition to do such investigations manually, the requirements to scientific accuracy make a program much better su conduct the experiments. We shall in this section outline how we can wr programs, often called *scripts*, for running other programs and archiv results.

Scientific investigation.

The purpose of the investigations is to explore the quality of nume solutions to an ordinary differential equation. More specifically, we the initial-value problem

$$u'(t) = -au(t), \quad u(0) = I, \quad t \in (0, T],$$

by the θ-rule:

$$u^{n+1} = \frac{1 - (1-\theta)a\Delta t}{1 + \theta a\Delta t} u^n, \quad u^0 = I\,.$$

This scheme corresponds to well-known methods: $\theta = 0$ gives the For Euler (FE) scheme, $\theta = 1$ gives the Backward Euler (BE) scheme, $\theta = \frac{1}{2}$ gives the Crank-Nicolson (CN) or midpoint/centered scheme.

For fixed I, a, and T, we run the three schemes for various valu Δt, and present in a report the following results:

1. visual comparison of the numerical and exact solution in a plot for each Δt and $\theta = 0, 1, \frac{1}{2}$,

2. a table and a plot of the norm of the numerical error versus Δt for $\theta = 0, 1, \frac{1}{2}$.

The report will also document the mathematical details of the problem under investigation.

.1 Available software

ppropriate software for implementing (5) is available in a program `model.py`[12], hich is run as

```
erminal> python model.py --I 1.5 --a 0.25 --T 6 --dt 1.25 0.75 0.5
```

he command-line input corresponds to setting $I = 1.5$, $a = 0.25$, $T = 6$, and ın three values of Δt: 1.25, 0.75, ad 0.5.

The results of running this `model.py` command are text in the terminal indow and a set of plot files. The plot files have names `M_D.E`, where `M` denotes ıe method (`FE`, `BE`, `CN` for $\theta = 0, 1, \frac{1}{2}$), `D` the time step length (here `1.25`, `0.75`, r `0.5`), and `E` is the plot file extension `png` or `pdf`. The text output looks like

```
0.0   1.25:    5.998E-01
0.0   0.75:    1.926E-01
0.0   0.50:    1.123E-01
0.0   0.10:    1.558E-02
0.5   1.25:    6.231E-02
0.5   0.75:    1.543E-02
0.5   0.50:    7.237E-03
0.5   0.10:    2.469E-04
1.0   1.25:    1.766E-01
1.0   0.75:    8.579E-02
1.0   0.50:    6.884E-02
1.0   0.10:    1.411E-02
```

he first column is the θ value, the next the Δt value, and the final column epresents the numerical error E (the norm of discrete error function on the ıesh).

.2 Required new results

he results we need for our investigations are slightly different than what is irectly produced by `model.py`:

1. We need to collect all the plots for one numerical method (FE, BE, CN) in a single plot. For example, if 4 Δt values are run, the summarizing plot

[12] `http://hplgit.github.io/teamods/writing_reports/doconce_src/model.py`

for the BE method has 2×2 subplots, with the subplot correspon the largest Δt is in the upper left corner and the smallest is in the right corner.

2. We need to have a table containing Δt values in the first colu the numerical error E for $\theta 0, 0.5, 1$ in the next three columns. Th should be available as a standard CSV file.

3. We need to plot the numerical error E versus Δt in a log-log plot

Consequently, we need to write a program (script) that can run `mode` described and produce the results 1-3 above. This requires combining ı plot files into a file and interpreting the output from `model.py` as c plotting and file storage.

If the script's name is `exper1.py`, we run it with the desired Δt v positional command-line arguments:

```
Terminal> python exper1.py 0.5 0.25 0.1 0.05
```

This run will then generate eight plot files: `FE.png` and `FE.pdf` summari plots with the FE method, `BE.png` and `BE.pdf` with the BE method, `CN.` `CN.pdf` with the CN method, and `error.png` and `error.pdf` with the plot of the numerical error versus Δt. In addition, the table with nu errors is written to a file `error.csv`.

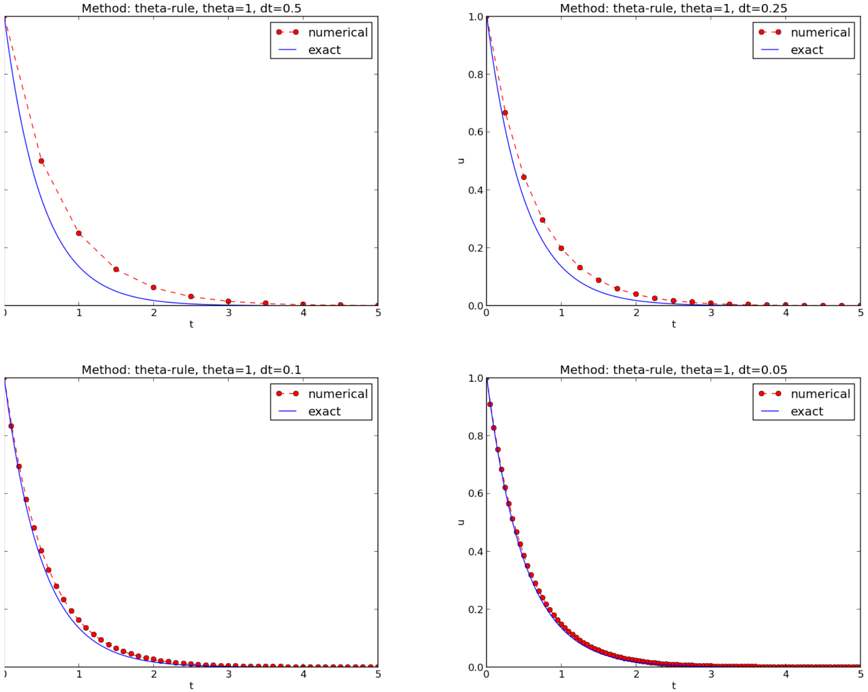
Reproducible science.

A script that automates running our computer experiments will ensure the experiments can easily be rerun by ourselves or others in the fu either to check the results or redo the experiments with other input Also, whatever we did to produce the results is documented in every c in the script. Automating scripts are therefore essential to makin research *reproducible*, which is a fundamental principle in science.

5.3 Combining plot files

Image files can be combined to new files using the `montage`[13] and `co` programs in the ImageMagick software suite. However, these programs suited for PNG files. For vector plots in PDF format one needs other preserve the quality: `pdftk`, `pdfnup`, and `pdfcrop`. Suppose you have f `f1.png`, `f2.png`, `f3.png`, and `f4.png` and want to combine them into table of subplots in a new file `f.png`, see Figure 3 for an example.

[13] `http://www.imagemagick.org/script/montage.php`
[14] `http://www.imagemagick.org/script/convert.php`



igure 3: Illustration of the Backward Euler method for four time step values.

The appropriate ImageMagick commands are

```
erminal> montage -background white -geometry 100% -tile 2x \
         f1.png f2.png f3.png f4.png f.png
erminal> convert -trim f.png f.png
erminal> convert f.png -transparent white f.png
```

he first command mounts the four files in one, the next `convert` command emoves unnecessary surrounding white space, and the final `convert` command akes the white background transparent.

High-quality montage of PDF files `f1.pdf`, `f2.pdf`, `f3.pdf`, and `f4.pdf` into .pdf goes like

```
erminal> pdftk f1.pdf f2.pdf f3.pdf f4.pdf output tmp.pdf
erminal> pdfnup --nup 2x2 --outfile tmp.pdf tmp.pdf
erminal> pdfcrop tmp.pdf f.pdf
erminal> rm -f tmp.pdf
```


5.4 Running a program from Python

Suppose you want to run some operating system command stored in `cmd`. For example, `cmd` could be `python model.py --I 1 --dt 0.5 0` following Python code executes `cmd` and loads the text output into `output`:

```
from subprocess import Popen, PIPE, STDOUT
p = Popen(cmd, shell=True, stdout=PIPE, stderr=STDOUT)
output, dummy = p.communicate()

# Check if run was successful
failure = p.returncode
if failure:
    print 'Command failed:', cmd; sys.exit(1)
```

In our case, we need to interpret the contents of `output` and store t in an appropriate data structure. Since the content is basically a table be transformed to a spread sheet format, we let the columns in the t lists, and we collect the columns in a dictionary whose keys are natural names: `dt` and the three values of θ. The following code translates `ou` such a dictionary of lists:

```
errors = {'dt': dt_values, 1: [], 0: [], 0.5: []}
for line in output.splitlines():
    words = line.split()
    if words[0] in ('0.0', '0.5', '1.0'):  # line with E?
        # typical line: 0.0   1.25:    7.463E+00
        theta = float(words[0])
        E = float(words[2])
        errors[theta].append(E)
```

5.5 The automating script

Running `model.py` for a set of Δt values and producing files as describe can be done by the following code:

```
import os, sys, glob
import matplotlib.pyplot as plt

def run_experiments(I=1, a=2, T=5):
    # The command line must contain dt values
    if len(sys.argv) > 1:
        dt_values = [float(arg) for arg in sys.argv[1:]]
    else:
        print 'Usage: %s dt1 dt2 dt3 ...' %  sys.argv[0]
        sys.exit(1)  # abort

    # Run module file and grab output
    cmd = 'python model.py --I %g --a %g --T %g' % (I, a, T)
    dt_values_str = ' '.join([str(v) for v in dt_values])
    cmd += ' --dt %s' % dt_values_str
    print cmd
```

```
    from subprocess import Popen, PIPE, STDOUT
    p = Popen(cmd, shell=True, stdout=PIPE, stderr=STDOUT)
    output, dummy = p.communicate()
    failure = p.returncode
    if failure:
        print 'Command failed:', cmd; sys.exit(1)

    errors = {'dt': dt_values, 1: [], 0: [], 0.5: []}
    for line in output.splitlines():
        words = line.split()
        if words[0] in ('0.0', '0.5', '1.0'):  # line with E?
            # typical line: 0.0   1.25:    7.463E+00
            theta = float(words[0])
            E = float(words[2])
            errors[theta].append(E)

    # Find min/max for the axis
    E_min = 1E+20; E_max = -E_min
    for theta in 0, 0.5, 1:
        E_min = min(E_min, min(errors[theta]))
        E_max = max(E_max, max(errors[theta]))

    plt.loglog(errors['dt'], errors[0], 'ro-')
    #plt.hold('on')  # Matlab style...
    plt.loglog(errors['dt'], errors[0.5], 'b+-')
    plt.loglog(errors['dt'], errors[1], 'gx-')
    plt.legend(['FE', 'CN', 'BE'], loc='upper left')
    plt.xlabel('log(time step)')
    plt.ylabel('log(error)')
    plt.axis([min(dt_values), max(dt_values), E_min, E_max])
    plt.title('Error vs time step')
    plt.savefig('error.png')
    plt.savefig('error.pdf')
    # Write out a table in CSV format
    f = open('error.csv', 'w')
    f.write(r'$\Delta t$,$\theta=0$,$\theta=0.5$,$\theta=1$' + '\n')
    for _dt, _fe, _cn, _be in zip(
        errors['dt'], errors[0], errors[0.5], errors[1]):
        f.write('%.2f,%.4f,%.4f,%.4f\n' % (_dt, _fe, _cn, _be))
    f.close()

    # Combine images into rows with 2 plots in each row
    image_commands = []
    for method in 'BE', 'CN', 'FE':
        pdf_files = ' '.join(['%s_%g.pdf' % (method, dt)
                              for dt in dt_values])
        png_files = ' '.join(['%s_%g.png' % (method, dt)
                              for dt in dt_values])
        image_commands.append(
            'montage -background white -geometry 100%' +
            ' -tile 2x %s %s.png' % (png_files, method))
        image_commands.append(
            'convert -trim %s.png %s.png' % (method, method))
        image_commands.append(
            'convert %s.png -transparent white %s.png' %
            (method, method))
        image_commands.append(
            'pdftk %s output tmp.pdf' % pdf_files)
        num_rows = int(round(len(dt_values)/2.0))
        image_commands.append(
            'pdfnup --nup 2x%d --outfile tmp.pdf tmp.pdf' % num_rows)
```



```
        image_commands.append(
            'pdfcrop tmp.pdf %s.pdf' % method)

    for cmd in image_commands:
        print cmd
        failure = os.system(cmd)
        if failure:
            print 'Command failed:', cmd; sys.exit(1)

    # Remove the files generated above and by model.py
    from glob import glob
    filenames = glob('*_*.png') + glob('*_*.pdf') + glob('tmp*.p
    for filename in filenames:
        os.remove(filename)

if __name__ == '__main__':
    run_experiments(I=1, a=2, T=5)
    plt.show()  # at the end of the program
```

We may comment upon many useful constructs in this script:

- `[float(arg) for arg in sys.argv[1:]]` builds a list of real r from all the command-line arguments.
- `failure = os.system(cmd)` runs an operating system comma another program. The execution is successful only if `failure` is z
- Unsuccessful execution usually makes it meaningless to continue gram, and therefore we abort the program with `sys.exit(1)`. A ment different from 0 signifies to the computer's operating system program stopped with a failure.
- `['%s_%s.png' % (method, dt) for dt in dt_values]` builds filenames from a list of numbers (`dt_values`).
- All `montage`, `convert`, `pdftk`, `pdfnup`, and `pdfcrop` commands for composite figures are stored in a list and later executed in a loop.
- `glob('*_*.png')` returns a list of the names of all files in the directory where the filename matches the Unix wildcard notation[15] * (meaning any text, underscore, any text, and then `.png`).
- `os.remove(filename)` removes the file with name `filename`.

5.6 Making a report

The results of running computer experiments are best documented in report containing the problem to be solved, key code segments, and t from a series of experiments. At least the part of the report contain plots should be automatically generated by the script that performs t

[15] http://en.wikipedia.org/wiki/Glob_(programming)

xperiments, because in that script we know exactly which input data that were sed to generate a specific plot, thereby ensuring that each figure is connected) the right data. Take a look at an example at `http://hplgit.github.io/eamods/writing_reports/sphinx-cloud/` to see what we have in mind.

'lain HTML. Scientific reports can be written in a variety of formats. Here e begin with the HTML[16] format which allows efficient viewing of all the expernents in any web browser. The program `exper1_html.py`[17] calls `exper1.py`) perform the experiments and then runs statements for creating an HTML file ith a summary, a section on the mathematical problem, a section on the numeral method, a section on the `solver` function implementing the method, and a ection with subsections containing figures that show the results of experiments here Δt is varied for $\theta = 0, 0.5, 1$. The mentioned Python file contains all the etails for writing this HTML report[18]. You can view the report on `http://plgit.github.io/teamods/writing_reports/_static/report_html.html`.

[TML with MathJax. Scientific reports usually need mathematical forulas and hence mathematical typesetting. In plain HTML, as used in the `xper1_html.py` file, we have to use just the keyboard characters to write matheatics. However, there is an extension to HTML, called MathJax[19], which allows rmulas and equations to be typeset with LaTeX syntax and nicely rendered in eb browsers, see Figure 4. A relatively small subset of LaTeX environments is ipported, but the syntax for formulas is quite rich. Inline formulas are look like `( u'=-au \)` while equations are surrounded by `$$` signs. Inside such signs, one an use `\[ u'=-au \]` for unnumbered equations, or `\begin{equation}` and `end{equation}` surrounding `u'=-au` for numbered equations, or `\begin{align}` nd `\end{align}` for multiple aligned equations. You need to be familiar with iathematical typesetting in LaTeX[20].

The file `exper1_mathjax.py`[21] contains all the details for turning the previous lain HTML report into web pages with nicely typeset mathematics. The orresponding HTML code[22] be studied to see all details of the mathematical ypesetting.

ATEX. The *de facto* language for mathematical typesetting and scientific eport writing is LaTeX[23]. A number of very sophisticated packages have been dded to the language over a period of three decades, allowing very fine-tuned ayout and typesetting. For output in the PDF format[24], see Figure 5 for an

[16] `http://en.wikipedia.org/wiki/HTML`
[17] `http://hplgit.github.io/teamods/writing_reports/report_generation/exper1_html.py`
[18] `http://hplgit.github.io/teamods/writing_reports/_static/report_html.html.html`
[19] `http://www.mathjax.org/`
[20] `http://en.wikibooks.org/wiki/LaTeX/Mathematics`
[21] `http://hplgit.github.io/teamods/writing_reports/report_generation/exper1_html.py`
[22] `http://hplgit.github.io/teamods/writing_reports/_static/report_mathjax.html.html`
[23] `http://en.wikipedia.org/wiki/LaTeX`
[24] `http://hplgit.github.io/teamods/writing_reports/_static/report.pdf`

We address the initial-value problem

$$u'(t) = -au(t), \quad t \in (0, T],$$
$$u(0) = I,$$

where a, I, and T are prescribed parameters, and $u(t)$ is the unknown function to be estimated. This mathematical model is relevant for physical phenomena featuring exponential decay in time.

Numerical solution method

We introduce a mesh in time with points $0 = t_0 < t_1 \cdots < t_N = T$. For simplicity, we assume constant spacing Δt between the mesh points: $\Delta t = t_n - t_{n-1}$, $n = 1, \ldots, N$. Let u^n be the numerical approximation to the exact solution at t_n. The θ-rule is used to solve (1) numerically:

$$u^{n+1} = \frac{1 - (1-\theta)a\Delta t}{1 + \theta a \Delta t} u^n,$$

for $n = 0, 1, \ldots, N-1$. This scheme corresponds to

- The Forward Euler scheme when $\theta = 0$
- The Backward Euler scheme when $\theta = 1$
- The Crank-Nicolson scheme when $\theta = 1/2$

Implementation

The numerical method is implemented in a Python function:

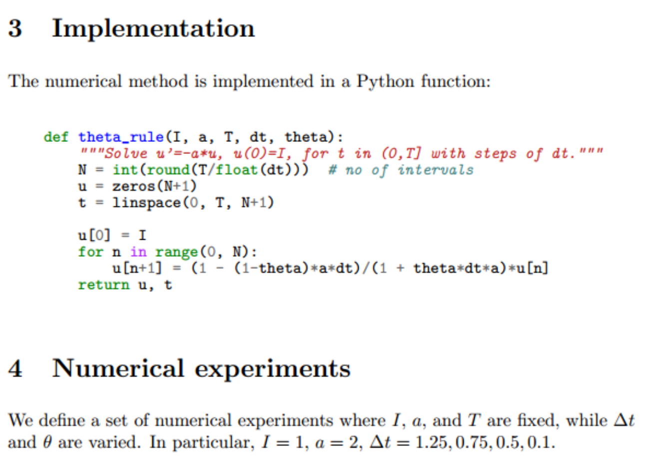
```
def theta_rule(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    N = int(round(T/float(dt)))  # no of intervals
    u = zeros(N+1)
    t = linspace(0, T, N+1)
```

Figure 4: Report in HTML format with MathJax.

example, LaTeX is the definite choice when it comes to quality. Th language used to write the reports has typically a lot of commands ir backslashes and braces[25]. For output on the web, using HTML (and not t directly in the browser window), LaTeX struggles with delivering high typesetting. Other tools, especially Sphinx, give better results and c produce nice-looking PDFs. The file `exper1_latex.py`[26] shows how to g the LaTeX source from a program.

Sphinx. Sphinx[27] is a typesetting language with similarities to HTI LaTeX, but with much less tagging. It has recently become very pop software documentation and mathematical reports. Sphinx can utilize L mathematical formulas and equations (via MathJax or PNG images). U nately, the subset of LaTeX mathematics supported is less than in full M (in particular, numbering of multiple equations in an `align` type enviror not supported). The Sphinx syntax[28] is an extension of the reStructu language. An attractive feature of Sphinx is its rich support for fancy la web pages[29]. In particular, Sphinx can easily be combined with variou *themes* that give a certain look and feel to the web site and that offers contents, navigation, and search facilities, see Figure 6.

[25] `http://hplgit.github.io/teamods/writing_reports/_static/report.tex.htm`
[26] `http://hplgit.github.io/teamods/writing_reports/report_generation/expe`
[27] `http://sphinx.pocoo.org/`
[28] `http://hplgit.github.io/teamods/writing_reports/_static/report_sphinx.`
[29] `http://hplgit.github.io/teamods/writing_reports/_static/sphinx-cloud/i`



3 Implementation

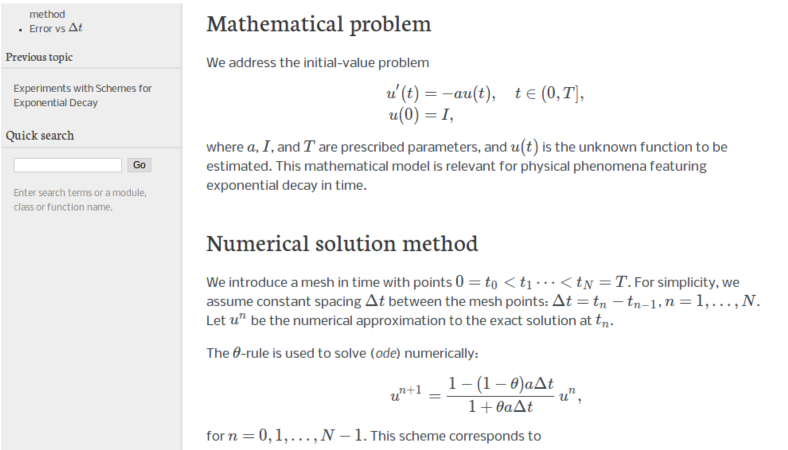
The numerical method is implemented in a Python function:

```
def theta_rule(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    N = int(round(T/float(dt)))  # no of intervals
    u = zeros(N+1)
    t = linspace(0, T, N+1)

    u[0] = I
    for n in range(0, N):
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

4 Numerical experiments

We define a set of numerical experiments where I, a, and T are fixed, while Δt and θ are varied. In particular, $I = 1$, $a = 2$, $\Delta t = 1.25, 0.75, 0.5, 0.1$.

Figure 5: Report in PDF format generated from LaTeX source.



method
- Error vs Δt

Previous topic

Experiments with Schemes for Exponential Decay

Quick search

Go

Enter search terms or a module, class or function name.

Mathematical problem

We address the initial-value problem

$$u'(t) = -au(t), \quad t \in (0, T],$$
$$u(0) = I,$$

where a, I, and T are prescribed parameters, and $u(t)$ is the unknown function to be estimated. This mathematical model is relevant for physical phenomena featuring exponential decay in time.

Numerical solution method

We introduce a mesh in time with points $0 = t_0 < t_1 \cdots < t_N = T$. For simplicity, we assume constant spacing Δt between the mesh points: $\Delta t = t_n - t_{n-1}, n = 1, \ldots, N$. Let u^n be the numerical approximation to the exact solution at t_n.

The θ-rule is used to solve (*ode*) numerically:

$$u^{n+1} = \frac{1 - (1-\theta)a\Delta t}{1 + \theta a \Delta t} u^n,$$

for $n = 0, 1, \ldots, N-1$. This scheme corresponds to

Figure 6: Report in HTML format generated from Sphinx source.

Iarkdown. A recently popular format for easy writing of web pages is Mark-
own[30]. Text is written very much like one would do in email, using spacing and
pecial characters to naturally format the code instead of heavily tagging the
xt as in LaTeX and HTML. With the tool Pandoc[31] one can go from Markdown
o a variety of formats. HTML is a common output format, but LaTeX, epub,
ML, OpenOffice, MediaWiki, and MS Word are some other possibilities.

[30] `http://daringfireball.net/projects/markdown/`
[31] `http://johnmacfarlane.net/pandoc/`

Wiki formats. A range of wiki formats are popular for creating n
the web, especially documents which allow groups of people to edit a
content. Apart from MediaWiki[32] (the wiki format used for Wikipedi
formats have no support for mathematical typesetting and also limited t
displaying computer code in nice ways. Wiki formats are therefore less
for scientific reports compared to the other formats mentioned here.

DocOnce. Since it is difficult to choose the right tool or format for w
scientific report, it is advantageous to write the content in a format tha
translates to LaTeX, HTML, Sphinx, Markdown, and various wikis. DocC
such a tool. It is similar to Pandoc, but offers some special convenient feat
writing about mathematics and programming. The tagging is modest, son
between LaTeX and Markdown. The program `exper1_do.py`[34] demo
how to generate (and write) DocOnce code for a report.

Worked example. The HTML, LaTeX (PDF), Sphinx, and DocOnce
for the scientific report whose content is outlined above, are exemplifi
source codes and results at the web pages associated with this teaching r
`http://hplgit.github.io/teamods/writing_reports`.

5.7 Publishing a complete project

A report documenting scientific investigations should be accompanied by
software and data used for the investigations so that others have a possi
redo the work and assess the qualify of the results. This possibility is im
for *reproducible research* and hence reaching reliable scientific conclusio

One way of documenting a complete project is to make a directory tr
all relevant files. Preferably, the tree is published at some project hosting
Bitbucket or GitHub[35] so that others can download it as a tarfile, zipfile,
the files directly using the Git version control system. For the investi
outlined in Section 5.6, we can create a directory tree with files

```
setup.py
./src:
   model.py
./doc:
   ./src:
      exper1_mathjax.py
      make_report.sh
      run.sh
   ./pub:
      report.html
```

The `src` directory holds source code (modules) to be reused in other p
the `setup.py` builds and installs such software, the `doc` directory cont
documentation, with `src` for the source of the documentation and

[32] `http://www.mediawiki.org/wiki/MediaWiki`
[33] `https://github.com/hplgit/doconce`
[34] `http://hplgit.github.io/teamods/writing_reports/report_generation/expe`
[35] `http://hplgit.github.com/teamods/bitgit/html/`

eady-made, published documentation. The `run.sh` file is a simple Bash script sting the `python` command we used to run `exper1_mathjax.py` to generate ne experiments and the `report.html` file.

Exercises

roblem 1: Make a tool for differentiating curves

uppose we have a curve specified through a set of discrete coordinates (x_i, y_i), $= 0, \ldots, n$, where the x_i values are uniformly distributed with spacing Δx: $_i = \Delta x$. The derivative of this curve, defined as a new curve with points (x_i, d_i), an be computed via finite differences:

$$d_0 = \frac{y_1 - y_0}{\Delta x}, \tag{6}$$

$$d_i = \frac{y_{i+1} - y_{i-1}}{2\Delta x}, \quad i = 1, \ldots, n-1, \tag{7}$$

$$d_n = \frac{y_n - y_{n-1}}{\Delta x}. \tag{8}$$

) Write a function `differentiate(x, y)` for differentiating a curve with co-rdinates in the arrays `x` and `y`, using the formulas above. The function should eturn the coordinate arrays of the resulting differentiated curve.

) Write a test function for the function in a).

) Start with a curve corresponding to $y = \sin(\pi x)$ and $n + 1$ points in $[0, 1]$. pply `differentiate` four times and plot the resulting curve and the exact $= \sin \pi x$ for $n = 6, 11, 21, 41$.
ilename: `curvediff.py`.

roblem 2: Make solid software for the Trapezoidal rule

n integral

$$\int_a^b f(x)dx$$

an be numerically approximated by the Trapezoidal rule,

$$\int_a^b f(x)dx \approx \frac{h}{2}(f(a) + f(b)) + h\sum_{i=1}^{n-1} f(x_i),$$

here x_i is a set of uniformly spaced points in $[a, b]$:

$$h = \frac{b-a}{n}, \quad x_i = a + ih, \ i = 1, \ldots, n-1.$$

Somebody has used this rule to compute the integral $\int_0^\pi sin^2 x dx$:


```
from math import pi, sin
n = 20
h = pi/n
I = 0
for i in range(1, n):
    I += sin(i*h)**2
print I
```

a) The “flat” implementation above suffers from three serious flaws:

1. A general numerical algorithm (the Trapezoidal rule) is implemen specialized form where the formula for f is inserted directly into t for the general integration formula.

2. A general numerical algorithm is not encapsulated as a general f with appropriate parameters, which can be reused across a wide r applications.

3. The lazy programmer dropped the first terms in the general formu $\sin(0) = \sin(\pi) = 0$.

Write a function `trapezoidal` that fixes these flaws. Place the funct module `trapezoidal`.

b) Write a test function `test_trapezoidal`. Call the test function expl check that it works. Remove the call and run pytest on the module:

```
Terminal> py.test -s -v trapezoidal
```

Hint. Note that even if you know the value of the integral, you do n the error in the approximation produced by the Trapezoidal rule. Howe Trapezoidal rule will integrate linear functions, and piecewise linear fu with discontinuities at the x_i points, exactly (i.e., to machine precision). test function on such linear functions $f(x)$.

c) Add functionality for computing $\int_a^b f(x)dx$ by providing f, a, b, a positional command-line arguments:

```
Terminal> python trapezoidal.py 'sin(x)**2' 0 pi 20
```

Note that the `trapezoidal.py` must still be a valid module file, interpretation of command-line data and computation of the integral r performed from calls in a test block.

Hint. To translate a string formula on the command line, like `sin(x)**2`, into Python function, you can wrap a function declaration around the formula and in `exec` on the string to turn it into live Python code:

```
import math, sys
formula = sys.argv[1]
f_code = """
def f(x):
    return %s
""" % formula
exec(code, math.__dict__)
```

he result is the same as if we had hardcoded

```
def f(x):
    return sin(x)**2
```

n the program. Note that `exec` needs the namespace `math.__dict__`, i.e., all ames in the `math` module, such that it understands `sin` and other mathematical unctions. Similarly, to allow a and b to be `math` values like `pi`, do

```
 = eval(sys.argv[2], math.__dict__)
 = eval(sys.argv[2], math.__dict__)
```

) Write a test function for verifying the implementation of the reading data om the command line.
ilename: `trapezoidal.py`.

Problem 3: Implement classes for the Trapezoidal rule

We consider the same problem setting as in Problem 2. Make a module with a ass `Problem` representing the mathematical problem to be solved and a class `olver` representing the solution method. The rest of the functionality of the odule, including test functions and reading data from the command line, should e as in Problem 2. Filename: `trapezoidal_class.py`.

Problem 4: Write a doctest

ype in the following program and equip the `roots` function with a doctest:

```
import sys
# This sqrt(x) returns real if x>0 and complex if x<0
from numpy.lib.scimath import sqrt

def roots(a, b, c):
    """
    Return the roots of the quadratic polynomial
    p(x) = a*x**2 + b*x + c.
```



```
    The roots are real or complex objects.
    """
    q = b**2 - 4*a*c
    r1 = (-b + sqrt(q))/(2*a)
    r2 = (-b - sqrt(q))/(2*a)
    return r1, r2

a, b, c = [float(arg) for arg in sys.argv[1:]]
print roots(a, b, c)
```

Make sure to test both real and complex roots. Write out numbers with 1 or less. Filename: `doctest_roots.py`.

Problem 5: Write a nose test

Make a test function for the `roots` function in Problem 4. Filename: `test`

Exercise 6: Make use of a class implementation

Implement the `experiment_compare_dt` function from `decay.py` usi `Problem` and class `Solver` from Section 4. The parameters `I`, `a`, `T`, the name, and a series of `dt` values should be read from the command line. F `experiment_compare_dt_class.py`.

References

[1] H. P. Langtangen. *A Primer on Scientific Programming With Pytho* in Computational Science and Engineering. Springer, fourth edition

ndex